RL-TR-95-140
In-House Report
June 1995

(D) ERIC MANUAL

James H. Lawton and Brendon P. Fowler

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

JGK

RL-TR-95-140 has been reviewed and is approved for publication.

APPROVED: [illegible]

NORTHRUP FOWLER, III
Chief, Software Technology Division
Command, Control and Communications Directorate

FOR THE COMMANDER: [illegible]

JOHN A. GRANIERO
Chief Scientist
Command, Control and Communications Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE June 1995	3. REPORT TYPE AND DATES COVERED In-House Oct 91 - Mar 95

4. TITLE AND SUBTITLE (D) ERIC MANUAL	5. FUNDING NUMBERS PE - 62702F PR - 5581 TA - 27 WU - 65
6. AUTHOR(S) James H. Lawton and Brendon P. Fowler	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Rome Laboratory (C3CA) 525 Brooks Road Griffiss AFB NY 13441-4505	8. PERFORMING ORGANIZATION REPORT NUMBER RL-TR-95-140
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory (C3CA) 525 Brooks Road Griffiss AFB NY 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER N/A

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: James H. Lawton/C3CA/(315) 330-2973

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

ERIC is an object-oriented programming language designed for supporting the development of intelligent, discrete, event-driven simulations. ERIC was developed as part of an ongoing research effort at Rome Laboratory (RL) to build a new generation of knowledge-based simulations that support Battle Management studies. DERIC (Distributed ERIC) is an extension of the ERIC simulation language, which was developed to exploit the natural parallelism and reduce the execution time of simulations. DERIC is upwardly compatible with ERIC; any simulation written in ERIC will run in DERIC, with few or no modifications, and is in many ways identical to ERIC. DERIC utilizes many ERIC mechanisms and has several new specialized constructs designed to facilitate parallel processing. This report is a description of the ERIC and DERIC programming languages, and how to use them to develop simulations.

14. SUBJECT TERMS Simulation, declarative languages, distributed computer, AI	15. NUMBER OF PAGES 52
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Table Of Contents

Preface........2
Objects, Classes, and Inheritance........3
1.1 Classes........3
1.2 Instance Objects........8
Behaviors, Messages, and Methods........10
2.1 Behaviors........10
2.1.1 Defining Behaviors........10
2.1.2 Messages are Not Evaluated........11
2.1.3 Messages are Actually Patterns........11
2.1.4 Inheritance of Behaviors........12
2.1.5 Before and After Daemons........13
2.1.6 Wrappers........15
2.2 Methods........16
2.3 Behaviors vs. Methods........16
Predefined Behaviors........17
3.1 Making Instance Objects........17
3.2 Killing Objects........18
3.3 Defining Behaviors and Daemons........18
3.4 Print Functions........18
3.5 Recalling Information from Objects........19
3.6 Setting Attribute Values........21
3.7 Sending Messages to Attributes........21
3.8 Tracing and Recording Messages........22
The Clock........23
4.1 Scheduling Events........24
4.2 Advancing the Clock using Universal Time........25
4.3 Setting and Recalling the Current Universal Time........25
4.4 The Queueing Mechanism in More Detail........25
4.5 Objects Can Die........26
4.6 Plan Manipulation........27
4.7 Clearing the Event Queue........28
Miscellaneous Matters........29
5.1 ERIC Objects are CLOS Objects........29
5.2 Accessing Instance Variables........29
5.3 Some Useful Predicates........29
5.4 Matching Facility........30
DERIC........31
6.1 Initialization........31
6.2 Class and Object Creation........31
6.3 Object Mobility........33
6.4 Tracing........33
6.5 DERIC Time Flow........34
6.6 Threads and Subthreads........35
6.7 TELL and COASK........36
6.8 Deadlock........37
6.9 Debugging........39
6.10 Other ERIC/DERIC Differences........39
Further Readings........40
Behaviors Defined for System Objects........41
Index........44

Preface

ERIC is an object-oriented programming language designed for supporting the development of intelligent, discrete, event-driven simulations. ERIC was developed as part of an on-going research effort at the Rome Laboratory (RL) to build a new generation of knowledge-based simulations that support Battle Management studies. Object-oriented programming languages are designed to support the development and maintenance of large, complex software systems. These systems are composed of objects that have certain attributes and behaviors. Objects communicate with each other by message passing. The object-oriented paradigm is particularly useful for modeling and simulation because many real-world systems are composed of objects whose interactions can be represented by messages.

DERIC (Distributed ERIC) is an extension of the ERIC simulation language, which was developed to exploit the natural parallelism and reduce the execution time of simulations. DERIC is upwardly compatible with ERIC; any simulation written in ERIC will run in DERIC, with few or no modifications, and is in many ways identical to ERIC. DERIC utilizes many ERIC mechanisms and has several new specialized constructs designed to facilitate parallel processing. It retains the same style of object-oriented programming, and contains only three major differences from the original ERIC.

This report is a description of the ERIC and DERIC programming languages. It does not assume the reader is familiar with object-oriented programming or simulation; however, it does assume that the reader is familiar with Lisp.

Chapter 1
Objects, Classes, and Inheritance

Objects are software entities that model real-world things. They combine the properties of both data and procedures. Thus objects maintain a local state and are capable of performing computations using and/or modifying their local state.

The local state of an object consists of its attributes. Attributes describe characteristics of an object. For example, an object modeling an automobile might have color, engine size, and number of doors as its attributes.

An object performs computation through behaviors. A behavior is a piece of procedural code associated with a given object or group of objects. Each behavior is identified by a particular message, which will invoke the behavior when the message is received by an object. For example, an automobile object might be sent a message telling it to start its engine. Message passing is how objects in ERIC interact with each other and the user. Message passing is one of the more important characteristics of ERIC and will be examined in more detail in Chapter 2.

There are two types of objects in ERIC: class objects and instance objects. A class is a description of one or more similar objects. In other languages, such as Pascal, classes correspond to types. An instance is an actual member of a class. For example, *3* is an instance of the class integer. A class can be a member, or subclass, of a more general class. *Integers* are a subclass of *numbers*. The ability to build more specific classes (such as *integers*) out of a more general class (*numbers*, in this case) is a powerful abstraction mechanism. ERIC provides this mechanism via class inheritance. A subclass can inherit attributes and behaviors from one or more superclasses.

[For the rest of this manual, use of the term "instance object" will be restricted to only those members of a class that are not themselves classes. In the numbers example given above, *3* will be considered an instance object, but *integer* will not. Also, the terms class and class object will be used interchangeably.]

1.1 Classes

Classes are used to model the elements of a system being simulated by an ERIC program. Most systems can be decomposed into various real-world objects, and this decomposition is mapped onto ERIC objects. ERIC classes describe the types of objects in the system and how they behave, while instances model the actual objects in the system. Class objects should be used to organize and manage the objects in a simulation; they should not play an operational part.

Figure 1 shows the relationship between the classes *Something, Animal, Marine-Animal, Mammal,* and *Marine-Mammal.* This example will be referred to several times in the following discussion.

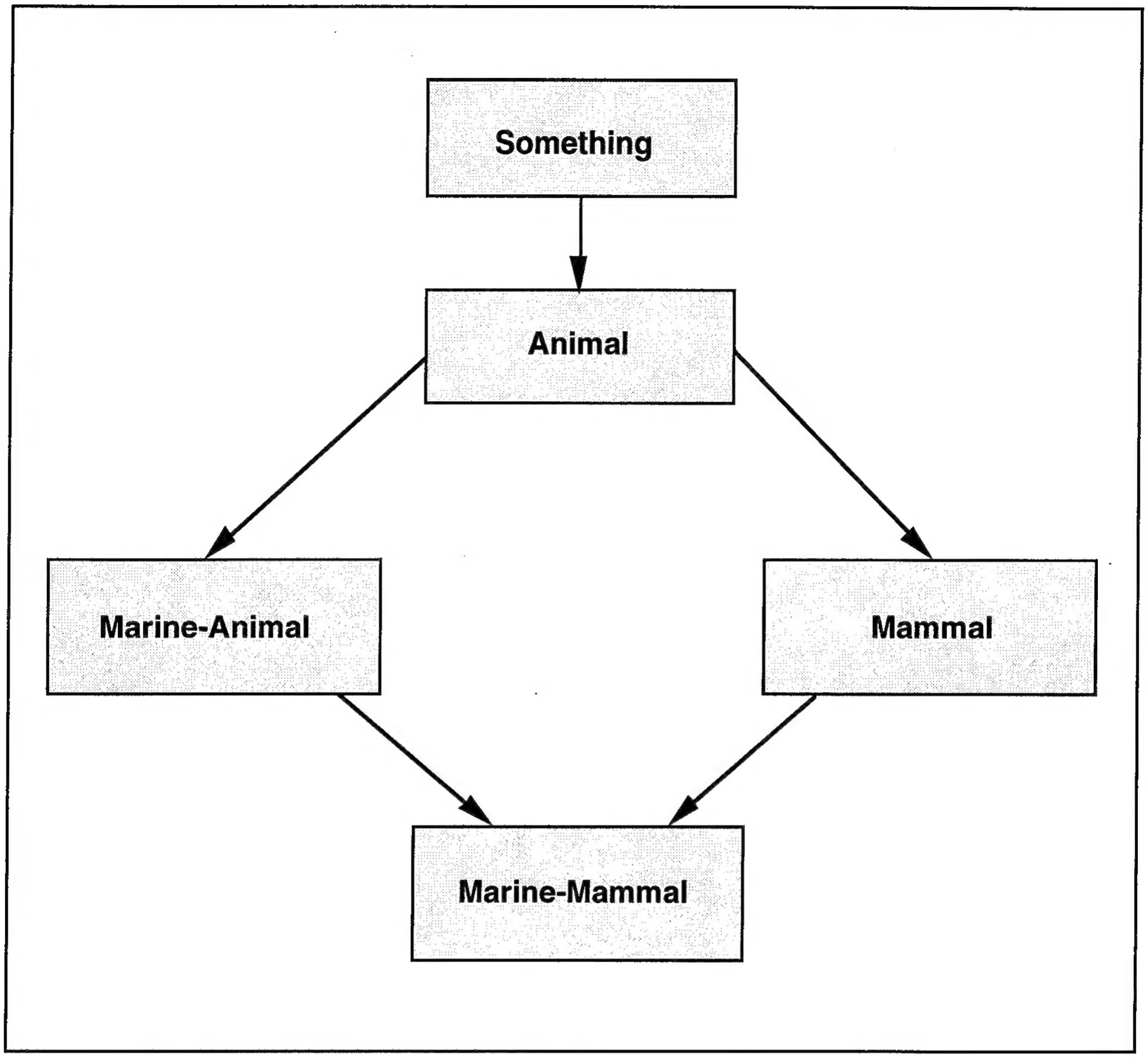

Figure 1. Example Hierarchy

The most primitive class in ERIC is called *something*. All other classes are built on top of (and hence, are subclasses of) *something*. *Something* provides many basic behaviors for all objects, such as how to print out attributes, make instances, and define new behaviors. New classes are defined in ERIC by using the *define-class* special form:

```
(define-class  <name>
   (:parents  <superclass>+)
   {(:documentation  <form>)}
   {(:class-attributes
      {<class-variable>*}
      {(<class-variable>   <init-value>)*})}
   {(:instance-attributes
      {<inst-variable>*}
      {(<inst-variable>   <init-value>)*})})
```

(Note: Items in parenthesis are required, items in braces are optional, * means zero or more are allowed, and + means one or more are allowed.)

The :parents list is the only required information for defining a new class (other than the class's name). It specifies the new class's inheritance chain, which determines the attributes and behaviors that will be inherited from other classes. In Figure 2, *animal* has *something* as the only superclass in its inheritance chain. *Marine-mammal*, however, has four inherited superclasses: *marine-animal, mammal*, *animal,* and *something*, in that order. Notice that *animal* and *something* are included in the inheritance chain even though they were not explicitly included in *marine-mammal's* :parents list.

```
(define-class Animal
  (:parents  something)
  (:class-attributes
    (status  'alive)))
```

```
(define-class Marine-Animal
  (:parents Animal)
  (:class-attributes
    (environment      'water)
    (movement         'swim)
    (breathing-organ  'gills))
  (:instance-attributes
    (length 10.0)
    (depth  nil)))
```

```
(define-class Mammal
  (:parents  Animal)
  (:class-attributes
    (environment       'land)
    (movement          'walk)
    (breathing-organ  'lungs))
  (:instance-attributes
    (length             72.0)
    (lung-capacity      nil)
    (gestation-period 270)))
```

```
(define-class  Marine-Mammal
  (:parents  Marine-Animal  Mammal)
  (:class-attributes
    (breathing-organ  'lungs))
  (:instance-attributes
    (length  50.0)))
```

Figure 2. Sample Class Definitions

Class attributes belong only to class objects; they are not inherited by instance objects. The :class-attributes list can be used to store any type of information, but is most useful for storing information that is common to all members belonging to a class and for managing instance objects and subclasses.

The optional :instance-attributes list declares what attributes an instance of the defined class will have in addition to those it inherits. The :instance-attributes list is composed of zero or more attribute names or name-value pairs, where the value is a Lisp form that is evaluated at the time of definition to a default initial value. If a default value is not provided, the attribute is set to nil. Attributes are inherited by first taking the union of the instance variables of each class in the inheritance chain and then eliminating duplicates. When eliminating duplicates, precedence is given to the leftmost attribute with a default value. (Precedence is leftmost with respect to the inheritance chain.) Figure 3 shows the class attributes and the instance attributes for the classes defined in Figure 2. In Figure 3, an instance of *marine-mammal* will have the following attributes: length (set explicitly to 50.0), depth, lung-capacity, and gestation-period (with an inherited value of 270 from Mammal). The optional :documentation list can be used to document the class being defined.

The order of the superclasses in the inheritance chain is very important, because this order determines which default attribute values and behaviors an object inherits.

Three rules govern the ordering of the inheritance chain:

1. A class always precedes its own superclasses.
2. The local ordering of superclasses in every object's :parents list is preserved.
3. Duplicate classes are removed from the ordering.

In Figure 2, the inheritance chain for *animal* is trivial: [*animal, something*]. The inheritance chain for *marine-animal* and *mammal* are [*marine-animal, animal, something*] and [*mammal, animal, something*], respectively. The one for *marine-mammal* is a bit more complex. Following the above three rules, we will construct *marine-mammal*'s inheritance chain. The chain begins with *marine-mammal* (Rule 1). Next, we add *marine-animal* and its inheritance chain. So far, the (incomplete) chain is: [*marine-mammal, marine-animal, animal, something.*]. The next class in the parents list is *mammal.* Upon examining the inheritance chain of *mammal*, we discover that *animal* and *something* are superclasses of *mammal.* According to Rule 1, we must place *mammal* before *animal* and *something.* The chain is now [*marine-mammal*, *marine-animal*, *mammal, animal, something.*] Continuing, we add the rest of the inheritance chain for *mammal*, which consists of *animal* and *something*, to the end of the partial chain. We now have: [*marine-mammal, marine-animal, mammal, something, animal, something*]. According to Rule 3, the duplicate *animal* and *something* are removed from the chain. The final result is the complete inheritance chain for *marine-mammal*:

[*marine-mammal, marine-animal, mammal, animal, something*]

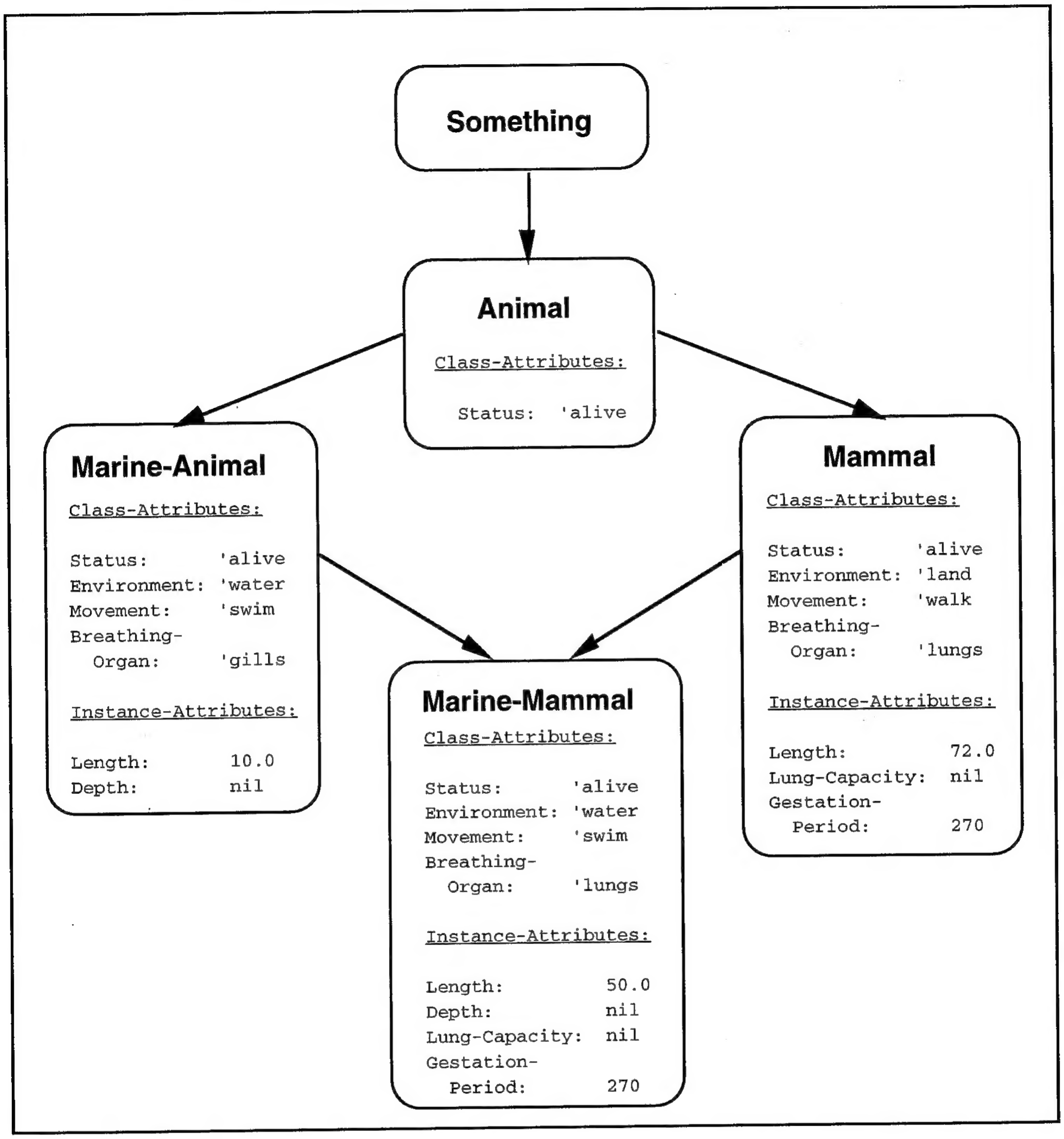

Figure 3. The Class and Instance Attributes of the Animal, Marine-animal, Mammal, and Marine-Mammal Classes.

It is possible to write class definitions that cannot be ordered using the three inheritance chain rules. As an example, consider these class definitions:

```
(define-class A (:parents B C))
(define-class C (:parents B))
```

From the definition of object *C*, we find that *C* must precede *B* because a class always precedes its superclasses, but *B* must precede *C* in order to preserve the local ordering of superclasses in the parents list of *A*. When an inheritance chain cannot be computed according to the three rules, ERIC signals an error and prints out information describing which classes are in conflict.

The value returned by the define-class form is the actual class object being defined. This object is also bound to the symbol in the name position of the define-class form. This symbol is declared to be a global variable when the define-class form is evaluated, so you can always use this symbol as a "handle" to access the class object anywhere in your program. You should be careful not to rebind this symbol to another value, or you may not be able to access the class object again.

Class objects identify themselves when printed as:

```
#<CLASS x>
```

where x is the name of the class, as specified in the class' define-class form. For example, after the Figure 2 class definitions have been evaluated, the result of typing the form

```
(eval marine-mammal)
```

at the top-level of a Lisp system, will be the class object *marine-mammal.* This object will print itself as:

```
#<CLASS MARINE-MAMMAL>
```

There is also a predicate function, *class-object?*, that will return true if its argument is an ERIC class object.

Every class object in ERIC has the following predefined attributes that may be inspected by users:

- print-name, the name of the class,
- parents, the :parents list from the class's definition,
- search-list, the list of the class's inheritance chain,
- offspring, a list of all the immediate subclasses and instances of this class, and
- documentation, the string that may have been included in the :documentation list of the class's definition.

Messages are passed to objects via the *ask* function. The message used to query a class for its attributes is:

(ask <class> recall your <attribute>)

Thus, (ask marine-mammal recall your parents) returns (#<CLASS MARINE-ANIMAL> #<CLASS MAMMAL>). The value of these attributes may be used in your programs, but you should never try to change their values. These attributes are established and maintained by ERIC; any modification by a user will probably cause problems in the evaluation of the program.

1.2 Instance Objects

Instances are the operational elements of an object-oriented simulation. It is the interaction of instance objects that simulates the interaction of real-world objects. Instance objects can be created in two ways: sending a class a *make instance message*, or by using the function *make-object*.

There are two types of *make instance message* commands: the first creates a named instance belonging to a specified class while the second creates an object, but also allows the specification of one or more attribute values. The first message is of the form:

(ask <class> make instance <name>)

which results in the creation of a named instance of class that is bound to the symbol <name>. As was the case with class names, the symbol <name> is declared to be a global variable and should not be rebound or you may lose access to the object. To aid in identification, the print name of the object created is #<name>.

The second make instance message is of the form

(ask <class> make instance <name> with {<attribute> <value>}*)

where <attribute> is an instance attribute of class and <value> is the value assigned to that attribute. For example:

```
(ask marine-animal make instance George with
    depth        72.0)  ; depth in inches
```

will return the object *George*, with print name #<GEORGE>. Any attributes that are not specified in the attribute-value portion of the message will take their normal default values. In this example, *George* will have the default length of 10.0 (see Figure 3).

The function *make-object,* of the form:

(make-object <class> &rest <attributes>*)

creates an anonymous instance of type <class>. An anonymous instance does not have a name associated with it, is not bound to any symbol, and prints itself as

#<unnamed instance of *class-name*: *N*>

where *class-name* is the name of the class and *N* is a unique serial number that can be used to visually tell if two anonymous objects are the same. The &rest arguments must be keyword-value pairs that specify the initial value of instance attributes. A keyword is simply an attribute name prefixed by a colon. Thus:

```
(make-object marine-animal
    :depth       72.0
    :length      10.0)
```

will create an anonymous instance of *marine-animal* with the same attribute values as *George*.

Every instance object in ERIC has the following predefined attributes that may be inspected by users:

- print-name, the name of the instance and
- parents, the :parents list from the object's class definition.

Chapter 2
Behaviors, Messages, and Methods

In Chapter 1, we said that objects are capable of performing computations and communicating with each other. In ERIC there are two mechanisms for doing this: behaviors and methods. Both of these mechanisms are a form of indirect function call, but while one emphasizes expressiveness, the other emphasizes efficiency.

2.1 Behaviors

A behavior is a piece of procedural code that is associated with a special sequence of symbols, called a message. Behaviors are invoked by sending messages to objects (this is often referred to as message passing). When an object receives a message, the procedure associated with the message is executed. Messages are passed to objects via the *ask* form. Borrowing from the examples in Chapter 1, *George*'s length could be determined by sending this message: (ask George recall your length) to which *George* would reply 10.0. The general form of the ask function is (ask <object> <message>), where <object> is either an ERIC class or instance object and <message> is the sequence of symbols denoting the message being sent to the object.

2.1.1 Defining Behaviors

Suppose we would like to provide members of the class *marine-animal* with a set of behaviors which might reflect how a *marine-animal* actually lives and behaves. A behavior for perpetuating the species could be defined by:

```
(ask marine-animal when receiving (perpetuate the species)
    (ask self find a marine-animal of opposite sex)
    (ask self perform courtship behaviors)
    (ask self settle down and raise kids))
```

There are several important things to notice in this example. First, we have just defined a behavior for members of the class *marine-animal* that is associated with the message "perpetuate the species." Behaviors are defined by a message of the form:

```
(ask <class> when receiving (<pattern>)
                        {<action>*})
```

where <pattern> is the message and <action> is zero or more Lisp forms that are to be evaluated when the behavior is invoked. Second, the use of *self* in the object position of the messages inside the behavior being defined requires an explanation. The variable *self* has a special meaning in ERIC. Each time a message is passed, *self* is bound to the object the message is being sent to. For example, during the evaluation of (ask George perpetuate the species), the variable *self* will be bound to the instance object *George*. *Self* allows the definition of behaviors that are applicable to any member of a class, without having to know the identity of the member. Third, when a member of *marine-animal* is sent this message, it sends out three messages to itself, which tell it to find a mate, perform courtship, and have children. The three messages sent out by our example must also have behaviors defined to handle them. ERIC behaviors are not limited to sending messages to *self* -- inside a behavior messages can be sent to any object, including both instances and classes.

The global variable **message-sender** may also be used in developing ERIC behaviors. This variable is bound to the object that sent the current message. If the message was not sent by an ERIC object, *message-sender* is bound to the symbol user.

2.1.2 Messages are Not Evaluated

Ask is a Lisp macro that evaluates only its object argument, the message argument is not evaluated. If you wish to include forms that get evaluated in a message, you must mark the forms with evaluation prefix characters. These special characters tell ERIC that the form directly following them should be evaluated, and the result should be spliced into the message sequence. There are two evaluation prefix characters in ERIC: the exclamation mark "!", and the ampersand "&". Both of these characters specify that the form immediately following them is to be evaluated, but the two characters differ in how they specify the result is to be spliced into the message sequence. The exclamation mark specifies that the result is to be spliced in "as is." The ampersand should only be used with forms that evaluate to a list; this list is then "unwrapped" and spliced into the message. Example 1 illustrates how the evaluation prefix characters work.

Example 1
Ask Form Evaluation

If	the-fish = George
and	place = Tahiti
then	(ask the-fish go to !place)
results in	(ask George go to Tahiti)
If	the-fish = George
and	place = (the islands)
then	(ask the-fish go to &place)
results in	(ask George go to the islands)
If	the-fish = George
then	(ask the-fish & '(swim upstream) !(+ 1 2) feet)
results in	(ask George swim upstream 3 feet)

2.1.3 Messages are Actually Patterns

Behaviors that could respond to classes of messages would be more useful than behaviors that can only respond to just one specific message. Take as an example the (ask George recall your length) message. Without some way to define behaviors for classes of messages, it would be necessary to write a different behavior for recalling every attribute an object has. This could be quite tedious. It is desirable to have a single behavior that would be responsible for handling all messages of the form "recall your x." This requires pattern matching.

Writing behaviors that handle classes of messages is accomplished in ERIC by using pattern variables in the message pattern of a behavior definition. The actual definition of the behavior for recalling the value of an object's attribute is:

```
(ask something when receiving (recall your >attribute)
    (object-get self attribute))
```

Attribute is a pattern variable in this definition. (The function *object-get* is explained in Chapter 5.) Pattern variables are denoted by the pattern matching prefix characters greater-than ">" and plus "x". When used in a pattern, these prefix characters act as wildcards, they match against anything. The ">" prefix will match single forms such as an atom or a parenthesized list. The "+" prefix will match any number of consecutive forms. If a symbol is prefixed by either of these characters in a pattern, the matcher will bind the symbol to the matched form. Example 2 shows multiple examples of how pattern matching prefix characters work. In the (recall your >attribute) example above, *attribute* is prefixed by the ">" character, so if a match occurs, *attribute* will be bound to the attribute whose value is being requested. Pattern variables perform the role of formal parameters for behaviors. This pattern matching technique provides a powerful mechanism for English-like messages. Example 3 shows the pattern matching capability in use.

Example 2
Matching Prefix Characters

Pattern	Datum	Results
(a b c)	(a b c)	match, no bindings
(a b c)	(a 1 c)	no match
(a >b c)	(a 1 c)	match, b = 1
(a >b c)	(a (1 2) c)	match, b = (1 2)
(a > c)	(a 1 c)	match, no bindings
(a >b c)	(a 1 2 c)	no match
(a +b c)	(a 1 2 c)	match, b = (1 2)
(a >b c)	(a c)	no match
(a +b c)	(a c)	match, b = ()
(a +)	(a 1 2 c)	match, no bindings
(>a +b c >d)	(e f c g)	match, a = e, b = (f), d = g

Example 3
Pattern Matching Example

If *Marine-Animal* has the behavior:

```
(ask Marine-Animal when receiving (move >direction >so-many feet +)
    (format t "~%  I have moved ~a feet ~a." so-many direction))
```

and *George* is sent:

```
(ask George move upstream 24 feet so you can see better)
```

George will respond with:

```
I have moved 24 feet upstream.
```

2.1.4 Inheritance of Behaviors

Objects inherit behaviors from their superclasses, just as they inherit attributes. The mechanism is fairly simple and is based on an object's inheritance chain. When an object is sent a message, it looks to its parent classes for a behavior with a pattern that matches the message. Each superclass on the inheritance chain is checked, from left to right, until a matching behavior is found. If no suitable behavior is found, ERIC signals an error. Going back to the objects defined in Figure 2, we'll evaluate the following behavior definitions:

```
(ask something when receiving (move)
    (print 'moving))

(ask marine-animal when receiving (move)
    (print 'swimming))

(ask mammal when receiving (move)
    (print 'walking))
```

After evaluation, *something*, *marine-animal*, and *mammal* will all have a pattern-action pair for the message "move" in their repertoire of behaviors. If we now evaluate (ask George move), the result is that “swimming” is printed. *George* looked to "his" parent, *marine-animal*, for a matching behavior, found it, and evaluated it. If we ask *marine-animal* or *mammal* to move, the result is that “moving” is printed. You might have expected *marine-animal* to print “swimming” and *mammal* to print “walking”, but remember that an object looks to its parent classes for its behaviors. The parent class for both *marine-animal* and *mammal* is *something*, so *something's* move behavior is evaluated. If we ask *marine-mammal* to move, “swimming” is printed. It may seem strange that a class object doesn't look to itself for a matching behavior, but there is a simple reason why it does not. In ERIC, a class defines a set of objects, it is not a member of that set. A class object is a member of its parent classes, so that is where it should look for its behaviors. Because all objects search for behaviors by looking to their parents, both instance objects and class objects have the same behavior inheritance mechanisms and act in a uniform manner.

2.1.5 Before and After Daemons

A daemon is a piece of code that is automatically invoked when some specified event occurs. In ERIC there are two kinds of daemons: before daemons, which are evaluated before a specific message is handled; and after daemons, which are evaluated after a specific message has been handled. Daemons are useful in a variety of ways. Often a behavior is defined for a given class, say *x*. Each of the subclasses of *x* would like to use this behavior, but a few of the subclasses need to perform different preparatory actions before this behavior is evaluated. Each subclass of *x* that needs to perform preparatory actions can define a before daemon to do them. Daemons are also useful for uncoupling input/output (I/O) functions, such as graphics, from behavior definitions. Because the I/O is separated from behaviors, it is easy to run simulations with or without graphics and to convert simulations to work with different I/O devices. Daemons are defined in a way similar to behaviors. Before daemons are defined by a message of the form:

```
(ask <class> before receiving (<pattern>)
                    {<action>*})
```

and after daemons are defined by the message:

```
(ask <class> after receiving (<pattern>)
                   {<action>*})
```

A class can have only one before and one after daemon for each message pattern. It is not necessary for a class to have a behavior that handles the original message, as long as one of its superclasses does.

The model for ERIC's message handling mechanism developed in Section 2.1.4 must now be modified to include daemons. When an object is sent a message, the parent classes on the object's inheritance chain are searched from <u>left to right for before daemons</u> that match the message. If a matching before daemon is found, it is evaluated and the search continues with the next class in the chain until all the superclasses on the inheritance chain have been searched. Then a search is made for the behavior which will handle the message (henceforth also called the primary behavior), which is made according to the description given in Section 2.1.4. After the primary behavior has been evaluated, the parent classes on the object's inheritance chain are searched from right to left for after daemons that match the message. If a matching after daemon is found, it is evaluated and the search continues with the next class in the chain until all the superclasses on the inheritance chain have been searched. Note that all the before and after daemons defined along an object's inheritance chain for a given behavior are executed, and that after daemons are executed in the reverse order of before daemons. Imagine that the inheritance chain is a path you are walking along. You start at the beginning of the path (the left end of the chain) and walk to the end of the path (the right end of the chain), stopping along the way to evaluate any appropriate before daemons you find. While you search for appropriate before daemons, you also look for the primary behavior to handle the message. When you come to the end of the path, you evaluate the primary behavior you found, and turn around to walk back to beginning of the path. As you walk back, you evaluate any appropriate after daemons you find as you come across them. This continues until you again reach the beginning of the path. An example of daemon inheritance is given below. Continuing in the spirit of Figure 2, suppose the following forms are evaluated:

```
(ask marine-animal before receiving (move)
    (print 'before-marine-animal-move))

(ask marine-animal after receiving (move)
    (print 'after-marine-animal-move))

(ask mammal before receiving (move)
    (print 'before-mammal-move))

(ask mammal after receiving (move)
    (print 'after-mammal-move))

(ask marine-mammal before receiving (move)
    (print 'before-marine-mammal-move))

(ask marine-mammal after receiving (move)
    (print 'after-marine-mammal-move))
```

If you now send *George* a message to move, the following would be printed:

```
before-marine-animal-move
swimming
after-marine-animal-move
```

A more complex example is to send an instance of *marine-mammal* a move message, which results in the following being printed:

```
before-marine-mammal-move
before-marine-animal-move
before-mammal-move
swimming
after-mammal-move
after-marine-animal-move
after-marine-mammal-move
```

Daemons may use parameter values that are accessible via pattern variables used in the pattern part of their defining message, but they do not return any values. They are useful for side-effect only. The value returned by a message pass is the value returned by the primary behavior, regardless of any daemons that may have been evaluated.

2.1.6 Wrappers

The final behavioral concept in ERIC that will be discussed is the wrapper. Daemons let you put code before and after the execution of a behavior; wrappers allow you to put code around the execution of a superclass's behavior. Wrappers are not a distinct class of procedural entities like before and after daemons, instead they are primary behaviors that use the *continue-passing* macro to continue searching the inheritance chain for message handlers. *Continue-passing* will only work in a behavior whose message is sent to an instance object. The *continue-passing* macro can be called either with no arguments or with one argument. In the no argument case, continue-passing allows the behavior currently handling a message to execute the next behavior along the inheritance chain capable of handling the original message. Consider the following simple example, based on the objects defined in Figure 2. Several new behaviors are added by sending the following messages:

```
(ask something when receiving (display >n)
    (print 'something)
    (princ n))

(ask marine-animal when receiving (display >n)
    (print 'marine-animal)
    (princ n)
    (continue-passing))

(ask mammal when receiving (display >n)
    (print 'mammal)
    (princ n))
```

After defining these new behaviors, we send *George* a message to print the number 5, (ask George display 5), and the resulting output is:

```
marine-animal 5
something 5
```

The "display 5" message was first handled by *marine-animal's* "display >n" behavior, which printed the first line of output, and then the call to continue-passing sends the "display 5" message on up the inheritance chain to *something*, which then prints the second line of output. Similarly, if Lenny, an instance of *marine-mammal*, is sent the same message, (ask Lenny display 5), the resulting output is:

```
marine-animal 5
mammal 5
```

because after *marine-animal*, the next superclass on *Lenny*'s inheritance chain that has a handler for the "display 5" message is *mammal*. The message sent up the inheritance chain by a wrapper can be altered by including a new message as an argument to *continue-passing*. If we redefine *marine-animal's* display behavior to be:

```
(ask marine-animal when receiving (display >n)
     (print 'marine-animal)
     (princ n)
     (continue-passing (display !(- n 1))))
```

and send the message (ask George display 5) the result now is:

```
marine-animal 5
something 4
```

Note that the display message was changed by calling the *continue-passing* macro with the new message to be sent up the inheritance chain as its argument. *Continue-passing* is a powerful mechanism for side-stepping normal message handling in ERIC. The new message passed up the inheritance chain can be anything you desire; it is not limited to being a variant of the original message. You should use this mechanism with discretion, however, because its run-time behavior may be difficult to comprehend by just looking at the code. Note that you can only use continue-passing at most once in a behavior. It will not work correctly if you try to use it more than once.

2.2 Methods

Methods are pieces of procedural code that are associated with a single symbol, and are invoked just like a function call. The methods in ERIC are actually CLOS methods, and the best reference for learning about them is the "Common Lisp, The Language (second edition)" by Guy Steele, so methods will not be explained in detail here. ERIC objects are CLOS objects, so all of the Lisp facilities that work on CLOS objects will also work on ERIC objects.

2.3 Behaviors vs. Methods

Why have both? Because each style has strengths and weaknesses. Behaviors provide a rich expressive capability that lets you encode knowledge in an easily understood format, but the pattern matching involved in message passing is expensive. Methods are efficient and fast, but are not very expressive. When should you use each? Methods should be used for mundane housekeeping chores and in places where speed is important. Behaviors should be used everywhere else. One should keep in mind, though, that in DERIC only behaviors are distributed automatically. If a programmer wishes a method to be defined in all environments, then the programmer will have to insure the method code gets loaded into each one.

Chapter 3
Predefined Behaviors

This chapter describes the predefined behaviors in ERIC. These behaviors have been defined for the object *something*, so every new class you define will inherit these behaviors. An alphabetical listing of these behaviors may be found in the section Behaviors Defined for System Objects at the end of this manual.

3.1 Making Instance Objects

To create a named instance object of a class, send that class the message:

(ask <class> make instance >ob)

The instance is then bound to the symbol specified by *ob*. See Section 1.2 for more information. Applicable only to class objects; instance objects will signal an error if sent this message. The command:

(ask <class> make instance >ob with +attributes)

does everything the previous message does, but allows you to assign initial values to the newly created instance's attributes. *Attributes* are one or more attribute-value pairs. See Section 1.2 for more details and an example. Applicable only to class objects; instance objects will signal an error if sent this message. If *ob* already exists as an instance object when either of these messages is sent, the old *ob* is "erased," in the sense that it is removed from its parent's list of offspring and its scheduled actions are canceled (see Chapter 4 for the significance of this action). However, the old *ob* does still exist somewhere in the Lisp environment, so that any pointers to it will still be intact. If *ob* already exists as a class object when either of these messages are sent, an error is signaled.

After an instance is created it is automatically sent the message:

(ask <object> initialize yourself)

The default behavior for this message does nothing except return the object. This message may be used for performing initialization actions by using after daemons.

In certain situations it may be desirable to have two instances of different classes sharing the same name, for instance the city New_York and the state New_York. To do this, use the command:

(ask <class> make plist instance >ob)

To access this type of instance, use the class name as a function accessor. For example, if city and state are defined as ERIC classes, the commands (ask city make plist instance New_York) and (ask state make plist instance New_York) will create two instance objects. To access the city New_York, use the accessor (get 'New_York 'city) or similarly (get 'New_York 'state) to access the state. Plist instances may also be created with attributes by using the command:

(ask <class> make plist instance >ob with +attributes)

3.2 Killing Objects

The command:

(ask <object> kill yourself)

"kills" an object by removing it from its parents' list of offspring and unbinds the object's name. All scheduled actions for this object are canceled (see Chapter 4). If a class object is killed, all of its descendants are also killed. To kill a class's instances, use the command:

(ask <class> kill all your instances)

If this message is sent to an instance, an error is signaled.

3.3 Defining Behaviors and Daemons

The following messages allow you to define primary behaviors and before and after daemons that are local to the class object receiving them:

(ask <class> when receiving >pattern +actions)
(ask <class> before receiving >pattern +actions)
(ask <class> after receiving >pattern +actions)

Pattern is the message template that will be associated with the *actions* code. See Section 2.1 for more details and an example. Applicable only to class objects; instance objects will signal an error if sent any of these messages.

The messages:

(ask <class> forget your local behaviors matching >pattern)
(ask <class> forget your local before daemons matching >pattern)
(ask <class> forget your local after daemons matching >pattern)

allow you to delete local primary behaviors and before and after daemons whose invoking message matches *pattern*. These behaviors return an integer telling the number of behaviors that were deleted. Applicable only to class objects; instance objects will signal an error if sent any of these messages.

3.4 Print Functions

One of the most common ways an object interacts with the user is to display information about its current state. There are several messages for telling an object to print various information. The following commands will have the object display all of its attributes and their values:

(ask <object> print yourself)
(ask <object> print your self)

If class objects are sent either of these messages the object will print its local behaviors, daemons, immediate descendants, and immediate parents. To have object display the values of all its class or instance attributes that were declared in the object's definition, use the command:

(ask <object> print your attributes)

This message is useful when you are only interested in seeing the attributes you gave the object, and don't care about the system defined attributes. The information printed by this message is less cluttered than that printed by the (print yourself) message.

To print the value of the attribute specified by the parameter *attribute,* use the command:

(ask <object> print your >attribute)

The command:

(ask <object> print your local messages)

prints the local behaviors and daemons defined on this object, in alphabetical order. If the object is a class, the behaviors and daemons that have been defined on the class are printed; if the object is an instance, the behaviors and daemons defined on its parent class are printed. To get a more extensive list of commands that may be sent to an object use the command:

(ask <object> print your messages)

This message is similar to the previous message, but it prints every message an object has a handler for. This includes behaviors and daemons that have been locally defined for an object and all those it inherits from its superclasses, except for the primary behaviors inherited from the superclass *something*. *Something* has so many behaviors that they clutter the screen and obscure the behaviors subsequently defined.

To print a specific message, use the command:

(ask <object> print your messages matching >pattern)

This will have the object print all of its behaviors and daemons capable of handling messages matching *pattern*. *Pattern* is a list which may contain wildcards as described in Chapter 2. For example, to see all the messages *something* can respond to that begin with the symbol print, you would send the message: (ask something print your messages matching (print +)).

3.5 Recalling Information from Objects.

Printing the values of attributes is good for human-object interaction, but not very useful for object-object interaction. The following behaviors return values that can be used by programs and other objects. To return the value of the attribute specified by the *attribute* parameter, use the command:

(ask <object> recall your >attribute)

The command:

(ask <object> recall the >attribute for your class)

returns the value for a class attribute, specified by the *attribute* parameter, of a class to which the object is a member. If the object sent the message is an instance, the class attribute will belong to its parent class. If the object is a class object, the attribute value returned will be from the first superclass on the object's parent list that has this attribute. If *attribute* cannot be found for a class, an error is signaled.

To return a list of the attributes defined for an object, use the command:

(ask <object> recall your attributes)

If the object is a class, its class attributes will be returned, if the object is an instance, the instance attributes will be returned.

To return a list of the local behaviors and daemons defined on the object, use the command:

(ask <object> recall your local messages)

If the object is a class, the behaviors defined for this class are returned; if the object is an instance, the behaviors defined on its parent class are returned. The more general command:

(ask <object> recall your messages)

returns a list of all the behaviors and daemons that are available to the object, including those which it has inherited from its superclasses.

To return a list of all the behaviors and daemons available to the object that handle messages matching *pattern* use the command:

(ask <object> recall your messages matching >pattern)

The descendants of an object may be obtained with the command:

(ask <class> recall your descendants)

which returns a list of all the descendants of a class. Descendants are defined to be the instances of a class and all of a class's subclasses and their instances. Applicable only to class objects, instance objects will signal an error if sent this message. The command:

(ask <class> recall your instances)

returns a list of all the instances belonging to this class or one of its subclasses. Applicable only to class objects, instance objects will signal an error if sent this message.

To return a list of all the subclasses of the object, use the command:

(ask <class> recall your subclasses)

This command is only applicable to class objects; instance objects will signal an error if sent this message.

The command:

(ask <class> recall your superclasses)

returns a list of all the superclasses of an object.

3.6 Setting Attribute Values

To set an attribute, use the command:

(ask <object> set your >attribute to >value)

This behavior assigns *value* to *attribute*. This works for both instance attributes and class attributes. If the object does not have *attribute* as one of its defined attributes, an error is signaled. To make assignments to an instance's class, use the command:

(ask <object> set the >attribute for your class to >value)

which assigns *value* to the class attribute specified by *attribute*. Applicable only to instance objects, if this message is sent to a class object, an error will be signaled. If the object's class does not have *attribute* as one of its defined class attributes, an error is signaled. The command:

(ask <object> add >x to your list of >attribute)

adds x to the list that is the value of *attribute*. If the object does not have *attribute* as one of its defined attributes, or if *attribute* does not contain a list, an error is signaled. To delete x from the list contained in *attribute,* use the command:

(ask <object> delete >x from your list of >attribute)

If the object does not have *attribute* as one of its defined attributes, or if *attribute* does not contain a list, an error is signaled. This behavior is destructive with respect to the list in *attribute*.

3.7 Sending Messages to Attributes

The command:

(ask <object> to ask each of your >attribute to +action)

sends the message *action* to each element of the list that is the value of *attribute*. For example, to have each of the offspring of *something* print itself, you would send this message: (ask something to ask each of your offspring to print your self).

To send the message *action* to the object that is the value of the *attribute,* use the command:

(ask <object> to ask your >attribute to +action)

If the command is to be sent to each of a class's instances, use the command:

(ask <class> to ask your instances to +action)

This will only work for class objects, if this message is sent to an instance, an error is signaled.

3.8 Tracing and Recording Messages

ERIC provides two ways to observe the passing of messages: tracing and recording. These facilities serve different purposes. The trace facility is useful for debugging programs and is similar to Lisp tracing facilities; the recorder saves a record of messages passed for later examination. The trace command is:

(ask <class> trace your messages matching >pattern)

This behavior traces all of a class object's primary behaviors that match *pattern*. *Pattern* may contain wildcard matching characters. When any object that is a member of this class receives messages that match *pattern*, the trace facility prints the following pieces of information to the current standard output stream: the current trace depth, the current simulation time, the class that is handling the message, the message itself, and the value returned by the invoked behavior. The trace facility indents nested message passes. This behavior is applicable only to class objects, if sent to an instance object, an error will be signaled. Also, messages that are handled by behaviors defined on *something* are not traced.

To trace all the messages sent to the class that are not handled by *something* behaviors, use the command:

(ask <class> trace all)

The command:

(ask <class> untrace your messages matching >pattern)

stops tracing all of the class object's primary behaviors whose invoking message matches *pattern*. This behavior is applicable only to class objects, if sent to an instance object, an error will be signaled. The

(ask <class> untrace all)

command stops tracing all messages sent to the class.

The recording facility is similar to the trace facility, except that the results are sent to a stream (e.g., file). Also, there is no indentation and only the message passes are recorded, not their results. The current simulation time and the object who sent the message are also included with each message record. This facility is accessed using the commands:

(ask <class> record your messages matching >pattern to >stream)
(ask <class> unrecord your messages matching >pattern)

Stream must already be open for output, and you must close it when you are finished recording. An example of a procedure using record is:

```
(with-open-file (record-output "file-name" :direction :output  :if-exists :overwrite)
    (ask marine-animal record your messages matching (+) to !record-output)
    (ask George move)
    (ask marine-animal unrecord your messages matching (+)))
```

Chapter 4
The Clock

The features of ERIC described so far have only dealt with its object-oriented programming capabilities. Objects alone do not make a simulation; there must also be a temporal control mechanism that allows objects to interact over time. In ERIC, this mechanism is provided by the clock object. The clock controls the flow of time in the simulation, allowing for actions to be scheduled for future execution. The clock is an instance object of class *simulation-clock*, and has three instance attributes: *simtime*, *event-list*, and *ticksize*. *Simtime* is the current time in the simulation. In ERIC, time is represented as an integer that has no built-in units. It is left to the programmer to decide what unit (if any) is associated with time, and to use this scale consistently within all object behaviors. For example, if you wish time to be measured in seconds, you should write behaviors that consistently refer to time in terms of seconds. The current value of *simtime* can be set and retrieved, respectively, using the following commands:

```
(ask clock set your simtime to >x)
(ask clock recall your simtime)
```

Simulations generally keep actions scheduled to happen in the future in some sort of time-ordered queue. In ERIC, this queue is maintained as the clock's *event-list*. The clock moves the simulation forward in time by executing events in the queue. The progression of simulation time is not smooth and continuous - it jumps from one event to the next. The interval of time between events never really exists.

The clock allows a simulation to be stopped at regular intervals or run for a specified period of time. The clock will start the execution of a simulation when it receives the message

```
(ask clock tick)
```

The simulation will then run for a length of time specified by the clock's *ticksize* attribute, which controls how many time units pass during each tick of the clock. Changing *ticksize* is one way to control how long a simulation runs. Another way is to use the message

```
(ask clock tick >n times)
```

which is simply a loop that sends clock a “tick message” *n* times. The "ideal" value of ticksize varies from simulation to simulation. Bear in mind that the value of *ticksize* has no direct effect on the behavior of the simulation; the clock's ticking is only for interaction and control purposes. Generally, the tick size should be longer than the mean time between events in a simulation. There is not much purpose in stopping every second of simulation time if the mean time between events is several minutes of simulation time. The last generic behavior for controlling simulation run length is

```
(ask clock run to completion)
```

which executes the simulation until there are no more events scheduled. If *simtime* is in seconds, the seconds/minutes/hours functions in Sections 4.1 through 4.3 may also be used.

4.1 Scheduling Events

There are two basic messages for scheduling actions to happen at some future time:

```
(ask clock to schedule >object to >action at >x)
(ask clock to schedule >object to >action in >x time units)
```

The first message schedules an action to happen at some absolute time *x*, which is an integer; the second message schedules an action to happen relative to the time at which the message was sent. *Action* must be a message that *object* can handle, or there will be an error signaled at some point in the future when the *object* tries to execute the action.

While a simulation may use any unit of time, it can be convenient for applications to speak in terms of hours, minutes, and seconds. To facilitate this several clock commands have been written which explicitly use these time units. The value stored in *simtime* is a Universal Time, which is the number of seconds elapsed since the beginning of 1900. The command to schedule an object to perform an action at a specific Universal Time is:

```
(ask clock to schedule >object to >action at time >x)
```

where *x* is a time-date string having a time, date, and optional time zone (e.g., "11/04/94 14:55:26 GMT"). *X* can also be the string "now".

The following commands may be used to schedule an object to perform an action at some delta time in the future:

```
(ask clock to schedule >object to >action in >x second)
(ask clock to schedule >object to >action in >x seconds)
(ask clock to schedule >object to >action in >x minute)
(ask clock to schedule >object to >action in >x minutes)
(ask clock to schedule >object to >action in >x hour)
(ask clock to schedule >object to >action in >x hours)
(ask clock to schedule >object to >action in >x minutes and >y seconds)
(ask clock to schedule >object to >action in >x hours and >y minutes)
(ask clock to schedule >object to >action in
                          >x hours >y minutes and >z seconds)
```

[Note: When we refer to "*x* seconds," we mean *x* seconds of simulated time. If there are no other messages scheduled on the clock, the message *action* will be sent immediately after *simtime* has been advanced.]

These scheduling commands may be used in object behaviors to simulate the advance of time. For instance, suppose we want a behavior that makes a member of the *marine-animal* class move every 10 seconds. This can be done by writing the behavior:

```
(ask marine-animal when receiving (move around)
   (ask self move)
   (ask clock to schedule !self to (move around) in 10 seconds))
```

This behavior first tells the object sent the message to move, and then schedules the object to move around again in ten seconds. We could set *George*, the *marine-animal* created in Chapter 1, in motion by sending it the message "move around" and then asking the clock to tick a few times.

Every ten seconds (of simulated time), *George* would move and then reschedule "himself" to move around in another ten seconds.

4.2 Advancing the Clock using Universal Time

In addition to the (ask clock tick) type commands, the clock can be advanced using minutes or hours. These commands are:

```
(ask clock run for >x minute)
(ask clock run for >x minutes)
(ask clock run for >x hour)
(ask clock run for >x hours)
```

These commands will tell the clock to tick (one ticksize at a time) until the requested amount of simulated time has transpired.

4.3 Setting and Recalling the Current Universal Time

The command to set the Universal Time is:

```
(ask clock set time to >x)
```

where *x* is a time-date string having a time, date, and optional time zone ((e.g., "11/04/94 14:55:26 GMT"). *X* can also be the string "now".

Commands are also provided to recall the current time and date. If the clock is set using (ask clock set time to "12/31/89 15:02:00"), then the current time and date can be returned using:

```
(ask clock what time is it)
        "15:02"
(ask clock what date is it)
        "12/31/89"
```

4.4 The Queueing Mechanism in More Detail

The event queueing mechanism is more involved than was described in Section 4.0. A thorough grasp of this mechanism is central to understanding how ERIC works. ERIC's scheduler is a bit unusual in that the event queue is distributed between the clock and the other objects in the simulation. Studies have shown that a simulation can spend a large percentage of its run time managing its event queue. Therefore, it is essential that a simulation language provides an efficient queueing mechanism. Managing the event queue consists primarily of two operations: inserting new events into their proper place in the ordered queue, and retrieving the event that should be executed next. Since the queue is time-ordered, the retrieval operation is trivial. Insertion is much more expensive than retrieval, with the cost of inserting an event depending on how large the queue is. It is also expensive to delete an event from the queue. However, in most simulations this is a rare operation, and some simulation languages do not even support event deletion. Like insertion, the cost of a deletion depends on the event queue's size.

ERIC was designed to support simulations that are composed of intelligent objects. One of the things intelligent objects in the real world often do is build, execute, and modify plans; it is reasonable to assume that intelligent objects in a simulation would need to do the same. Therefore, ERIC was designed to make it easy and efficient for objects to examine and modify their scheduled future events. This means providing an efficient way to search for, insert, and delete events in the event queue. This led to the development of ERIC's distributed queueing mechanism.

As mentioned earlier, the length of the event queue is an important factor in how long it takes to perform search, insert, and delete operations. Therefore, it is desirable to keep the event queue as small as possible. The first part of ERIC's queue mechanism, the clock's *event-list* attribute, keeps track of which objects have actions scheduled and when the actions are to be executed. Actually, the event-list only keeps pointers to a small number of the possibly many events scheduled for a given object. Ideally, the *event-list* will contain only one reference to a given object, but this is not always possible. Keeping only a few pointers to each object helps reduce the size of the *event-list.* Each object's *schedule* attribute contains information about its scheduled future actions. *Schedule* stores a time ordered list of data structures called plans, which contain three pieces of information: what action is to be performed, when the action is to be performed, and who scheduled the action. Allowing each object to store its own plans facilitates the examination and modification of plans. It is far more efficient to have an object rummage through its *schedule* attribute, which contains only its own scheduled future actions, than to have it rummage through a monolithic event queue that contains every object's future actions.

When an action is scheduled for object *X*, the clock checks *X's schedule* for plans. If *X* currently has no plans in its schedule, or if the new action is supposed to happen before any event currently in the schedule, the clock places a reference to *X's* future action in *event-list.* If *X* currently has actions scheduled to happen before the new action, then there already is an appropriate reference to *X* in the *event-list*, so a new reference is not inserted. (This is how the *event-list* size is kept down to about one reference per object.) In either case, the future action is inserted into the object's *schedule.*

When the clock pulls an event off *event-list* for execution, it checks the *schedule* attribute of the object specified in the event to see if the object has a plan that should be executed at the current time. If so, it is executed; if not, nothing happens. In either case, the clock then puts a reference to the object's next scheduled action into the event-list. The cost of checking an object's *schedule* when there is nothing to do is less expensive than trying to delete an action from the event queue. This allows plans to be deleted or modified efficiently.

4.5 Objects Can Die

Another benefit of the distributed event queue is that it is easy to "kill" an object. In the physical world, objects can be destroyed and are no longer able to perform actions. All instance objects in ERIC have an *eric-status* attribute which is either the symbol alive or dead. (You should avoid modifying eric-status in your own code.) An instance object can be killed by sending it the message

(ask object kill yourself)

When an object is killed, it is removed from its parents' list of offspring, its scheduled actions are canceled, and its name is unbound. However, the object still exists somewhere in the Lisp environment, and any pointers to the object that you may have stored in data structures still point to the now dead object. Dead ERIC objects are able to respond to messages sent to them, but they cannot be scheduled to perform any actions in the future. If you try to schedule an action for a dead object, an error will be signaled. It may be necessary in your application to perform certain actions when an object dies. This can be done using daemons or wrappers. Two Lisp predicate functions, live-object? and dead-object? (see Section 5.3), tell if an object is alive or dead, respectively.

4.6 Plan Manipulation

As mentioned earlier, the future scheduled actions for an object are stored in the object's *schedule* attribute in the form of plans. Each plan is a data structure that holds three pieces of information: the action to be performed, the time when the action should be performed, and who scheduled the action. These pieces of information can be retrieved from a plan with the functions *get-plan-action, get-plan-time,* and *get-plan-scheduler.* Each of these functions takes a plan as their sole argument.

Several behaviors are provided to manipulate an object's schedule. For instance:

(ask <object> recall your schedule

recalls all of an object's plans from its schedule. The command:

(ask <object> forget your plans matching >pattern)

removes all of an object's plans whose action matches *pattern* from its schedule. The matching plans will not get executed. *Pattern* may include any number of wildcard pattern variables. The command:

(ask <object> forget your plans at time >x)

forgets all of an object's plans that are scheduled to happen at time *x*. These plans will not get executed. The command:

(ask <object> forget your plans before time >x)

forgets all of an object's plans that are scheduled to happen before (but not including) time *x*. These plans will not get executed. The command:

(ask <object> forget your plans after time >x)

forgets all of an object's plans that are scheduled to happen after (but not including) time *x*. These plans will not get executed. The command

(ask <object> forget your plans between times >time1 and >time2)

forgets all of an object's plans that are scheduled to happen between times *time1* and *time2*, inclusive. These plans will not get executed.

If these behaviors do not satisfy the needs of your application, you can write your own. You are allowed only to remove plans from an object's schedule. If you wish to add plans, you must send scheduling messages to the clock. *Schedule* contains a list of plans, ordered from the earliest to latest time of scheduled execution. Remember to maintain this ordering when you modify *schedule*, or things will go awry.

4.7 Clearing the Event Queue

You can clear out the event queue and cancel all the scheduled events with the message

(ask clock cancel all events)

Chapter 5
Miscellaneous Matters

The purpose of this chapter is to present several short topics that do not fall neatly into the previous chapters. Most of these topics deal with Lisp functions that are useful for programming in ERIC.

5.1 ERIC Objects are CLOS Objects

ERIC objects are implemented on top of the Common Lisp Object System (CLOS). Any of the facilities provided for dealing with CLOS objects may be used with ERIC objects. All class objects in ERIC are CLOS objects of the type “standard-class class-object.” Instance objects are also CLOS objects, but they may come in a variety of types.

5.2 Accessing Instance Variables

The *define-class* form creates accessor functions for each of the class's instance attributes. Accessor functions perform the same job as the "recall your >attribute" and "set your >attribute to >value" messages, but accessor functions are much more efficient. In some instances you may choose to use one of the accessor functions instead of the ERIC messages, but remember that you loose the ability to record and use daemons. Inside of a behavior, you can reference an instance attribute by simply using its name, just as you can in a method. The automatically defined accessors take the form:

```
(<class-name>-<attribute> <object>)
```

For example, using the classes defined in Chapter 1:

```
(mammal-length George)
```

would return the value of the *length* attribute for the mammal *George*. Similarly, one would use the Lisp *setf* form to set the value of an accessor.

There is one more way to access instance variables, and that is with the functions *object-get* and *object-put*. They are of the form:

(object-get object attribute)
(object-put object attribute value)

Like the recall and set messages, these two functions are useful when the name of the attribute you wish to access is known only at run time, but they are more efficient than message passing.

5.3 Some Useful Predicates

This section presents some useful Lisp predicate functions. They have been presented separately elsewhere in this report, but are gathered here for convenience. All of the predicates take a single argument.

object?	Returns true if the argument is an ERIC object.
class-object?	Returns true if the argument is an ERIC class object.
instance-object?	Returns true if the argument is an ERIC instance object.
live-object?	Returns true if the argument is a live ERIC object.
dead-object?	Returns true if the argument is a dead ERIC object.

5.4 Matching Facility

The pattern matching facility of ERIC is available for use via the *ematch* function, of the form: (ematch pattern datum) which matches the datum against the pattern. *Pattern* may contain pattern variables and wildcards. There are two caveats when pattern matching: the same pattern variable should not appear twice in the same pattern, and pattern variables should not appear immediately after a + or +*var* wildcard. The result returned by *ematch* is one of three possibilities: nil, if there is no match; t, if the match is successful but there are no variables in the pattern; or an association list of variables and bindings, if the match is successful and there are variables in pattern.

Chapter 6
DERIC

DERIC (Distributed ERIC) is an extension of the ERIC simulation language (which you are now familiar with) which was developed to exploit the natural parallelism and reduce the execution time of simulations. DERIC is upwardly compatible with ERIC; any simulation written in ERIC will run in DERIC, with few or no modifications, and is in many ways identical to ERIC. DERIC utilizes many ERIC mechanisms and has several new specialized constructs designed to facilitate parallel processing. It retains the same style of object-oriented programming, and contains only three major differences from the original ERIC.

DERIC introduces the concept of an *environment*, which is simply a single Lisp image running on a computer. We use this term because it is possible to have multiple DERIC environments running on the same computer (although not with the current implementation).

6.1 Initialization

In order to run parallel simulations in DERIC, DERIC must be loaded into each evironment that will be used in the simulation. The different environments must then be made aware of each other. This is accomplished through the use of the *new-environment* command:

(new-environment '<env-name>)

This command is used to notify an individual environment about other environments and sets up the communication paths between them. For example, suppose we have two environments, Lestat and Armand. The following would be evaluated in the environment Lestat to make it aware of the environment Armand:

```
(new-environment 'armand)
```

In order to run the simulation smoothly, a single environment must be chosen as the main environment. It is in this environment that the system-objects will be consolidated, namely the objects *something*, *object-maker*, *tracer*, *user*, *simulation-clock*, *clock*, *master-clock*, *slave-clock*, and the *ticker*, which are discussed later. These objects reside in the main environment, and ensure that the same object is not in existance in two different evironments. The function:

(consolidate-my-environment '<env-name>)

will make the environment <env-name> the main environment.

6.2 Class and Object Creation

An important new object in DERIC is the *object-maker*. The *object-maker* is an object that combines the functionality of the *define-class* function and the "make instance" message used in ERIC. The *object-maker* defines new classes with the following message:

(ask object-maker make <name> a subclass of <parent>)

with <name> being the name of the new class. This message can be further expanded to optionally include class-attributes and instance-attributes:

```
(ask object-maker make <name> a subclass of <parent> with
    {class-attributes
        (<class-variable>  <init-value>)*}
    {instance-attributes
        (<inst-variable>  <init-value>)*}
    )
```

Taking the earlier example in ERIC for defining the class *marine-animal*:

```
(define-class marine-animal
    (:parents animal)
        (:instance-attributes (length 10.0) (depth nil)))
```

We can create the same class by sending the following message to the *object-maker*:

```
(ask object-maker make marine-animal a subclass of animal with
    instance-attributes
        (length 10.0) (depth nil))
```

with *animal* being a previously defined subclass of *something*. In order to create instance-attributes we use a similar request:

(ask object-maker make <name> a <parent>)

instead of the ERIC command:

```
(ask <class> make instance <name> with +attributes)
```

For example:

```
(ask object-maker make george a marine-animal with depth 10)
```

To define behaviors in DERIC we use the same structure as in ERIC. For example:

```
(ask george when receiving (swim) "glub glub")
```

Thus when we (ask george swim) he will return "glub glub".

The user can augment existing classes using the following message:

(ask object-maker remake <name> a <parent> with +attributes)

For the sake of readability, messages can use either "a" or "an" when creating classes and objects. For example, if we wish to make *sam* an instance-object of the class *Elephant* we can send either:

(ask object-maker make sam **a** elephant)

or

(ask object-maker make sam **an** elephant)

6.3 Object Mobility

Once objects and classes have been created via the *object-maker*, knowledge of them needs to be given to each separate environment, otherwise the environments will return error statements if queried about the status of these objects. The command *announce* is used to announce objects to the other environments. For example, if the class *marine-animal*, with its one offspring *George* was created in Armand, and we wish to tell Lestat about it so they can interact, we would evaluate the following message on Armand:

(ask marine-animal announce yourself to Lestat)

If we then asked about the class *marine-animal* on Lestat, it would return the statement <Armand's marine-animal>, meaning it knows about the class, but that the class still resides in Armand. At this point the environment on Lestat does not know about *marine-animal*'s offspring. When a class or instance-object is announced to a new environment, that environment learns about that class or object and its parents, but not about its offspring. Therefore the environment on Lestat would now know about the class *marine-animal*, but not about the instance-object *George*. We would then need to tell *George* to announce himself to the environment on Lestat. This process can be simplified by just telling *George* to announce himself to Lestat in the first place, because the environment on Lestat would then automatically learn about the class *marine-animal*, the class *animal*, and the class *something* because these are *George*'s parents.

Each object created has a *home environment*; the environment in which it was created. If *George* was created in Armand, Lestat would return <Armand's George> when asked because *George* was created there. Objects can be created in an environment and then moved to one of the other environments in the simulation. This keeps the user from having to physically go to each environment and create the objects for the simulation. For example, *George* can be asked to change his home environment by sending him the message:

(ask <object> migrate yourself to <environment>)

George would then switch his home environment to Lestat. If we check, Lestat's environment would return <George> when asked, while Armand's would now return <Lestat's George>, because Lestat is now *George*'s environment. The message:

(ask object-maker recall the object-maker of <environment>)

allows the user to get an *object-maker* instance that resides in a remote environment. This instance can be used to directly create objects in an environment via that environment's *object-maker*. For example:

(ask (ask object-maker recall the object-maker of Lestat) make Fred a fish)

would create an instance object of the class *fish* named *Fred* in Lestat's environment.

6.4 Tracing

DERIC uses the same *trace* and *record* functions previously defined in ERIC, but has one major additon called the *Tracer*. The *Tracer* is an object that is useful in debugging and combines the trace and record facilities in ERIC. When objects are asked to record or trace their messages, the output is handled by the *tracer* object, not by function calls as it is in ERIC. The actual tracing and recording messages are set up using most of the same messages as in ERIC.

For example if Armand and Lestat, our two environments, are running a simulation using *marine-animals* and we wish to see a trace of the behavior (swim away) we would send the message:

```
(ask marine-animal trace your messages matching (swim away))
```

The output would contain the simulation time, the class, the message itself, the value returned by the behavior, the trace depth, and the standard output stream. Each environment has its own local tracer, and if the previous message was sent in the environment Lestat, then the local *tracer* object would trace the message by default for all *marine-animal* objects, wherever they reside. If multiple trace messagess are sent to multiple environments then the specific *tracer* to which the output will be sent should be specified so that the output of each environment is not printed just on that environment. By channeling the trace messages back to a single *tracer* in a specific environment, we further enable the simulation to be controlled by a single environment overseeing the others, which was created using the previously defined *consolidate-my-environment* command. The message:

(ask tracer recall the tracer of <environment>)

with <environment> representing a specific environment (i.e. Lestat) allows the user to choose the tracer of the environment that the output of the subsequent trace messages will be sent to. The user can change which *tracer* will receive the messages with the message:

(ask <class> trace your messages matching <pattern> to <tracer-object>)

The ERIC message (ask <class> record your messages matching >pattern to >stream) is not present in DERIC. Instead, recording is done through the *tracer* objects, using the following two messages:

(ask <class> record your messages matching <pattern>)
(ask <class> record your messages matching <pattern> to <tracer-object>)

The first will use the *tracer* object of the local envrionment; the second is used to record through a *tracer* object in another environment. The *tracer* objects have instance varaibles *tracing-stream* and *recording-steam*, which are by default set to *t* (the standard output). Output can be redirected to windows or files by binding these variables to streams. For example, if we had a CLIM window in our environment bound to *win*, and an open file stream bound to *outfile*, we could redirect the tracing and recording output with:

```
(ask tracer set your tracing-stream to !*win*)
(ask tracer set your recording-stream to !*outfile*)
```

6.5 DERIC Time Flow

DERIC allows the user to choose from two different Clock objects: the original sequential Clock object, which is a direct port of ERIC's Clock object (discussed in Chapter 4), and a two-part clock referred to as the "Master-Slave Clock" (FIgure 4). The latter is one of the special modifications made to DERIC for use in parallel simulations. The "master" clock, called the Ticker, resides in a single special environment; each environment has its own slave clock, including the environment containing the Ticker. Although the Master-Slave Clock is implemented as several objects, it is considered a single object and referred to as the Clock. In addition the

Master-Slave Clock can be treated as a single object in that there exists only one simulation time and one future events list.

A typical "single clock tick" cycle starts with the Ticker sending out a message to all the slave-clock objects that reads "(it is now <simtime>)," where <simtime> is the current simulation time as considered by the Ticker. Each individual slave-clock executes any events that they have scheduled to occur at <simtime>. When each slave-clock finishes executing all the events scheduled for <simtime>, they each individually return a *next-event-time* to the Ticker, which is the simulation time of the earliest event currently on that specific clock's future events list. When all of the clocks have returned this message, the Ticker selects the smallest of the next-event-time's and sets the current simulation time to it, allowing the simulation to proceed and ensuring that no events are missed.

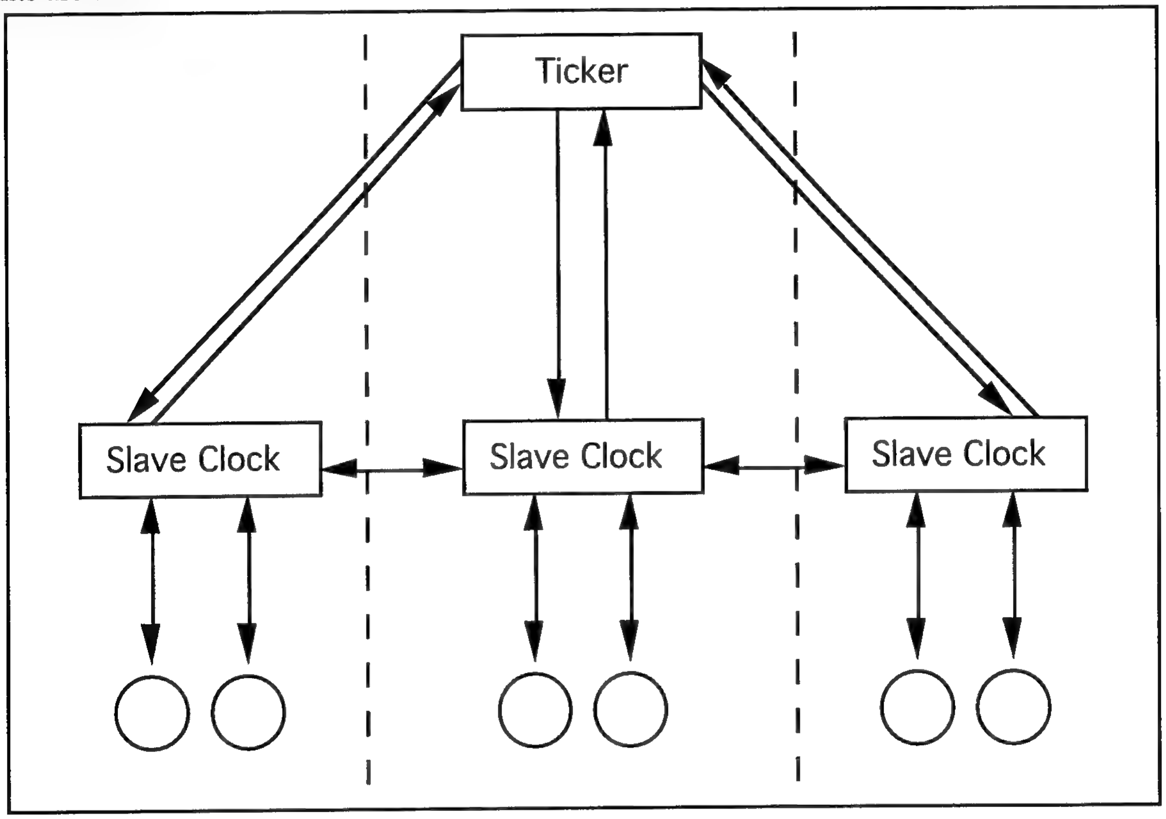

Figure 4. Master-Slave Clock

Each slave clock is referenced as *%Clock.* By default the slave clocks are not engaged when DERIC is first loaded and must be engaged by issuing the following commands in each environment:

(setq old-clock clock)

followed by

(setq clock %clock)

These slave clocks in turn communicate with the Master-clock, the Ticker, and allow for the parallel execution of events that are scheduled to occur at the same simulation time.

6.6 Threads and Subthreads

In DERIC, *threads of control* delineate the flow of execution through the objects. Threads are sequences of atomic actions that contain identities. Threads are created by *object requests* issued by either the user or from a system object such as the Clock. Object requests are messages sent to objects asking them to perform a certain behavior. A thread S is called a *subthread* of thread T if an object controlled by T issues a request creating S. Subthreads such as S inherit the permissions from their parent threads, and will cause execution to occur in parallel to its parent's thread.

For example, in Figure 5 thread t1 was created by a request from an object to object O1. Object O1 then makes two requests, one to object O4 which spawns the subthread t1.1, and the other to object O2, which continues the execution sequence of thread t1.

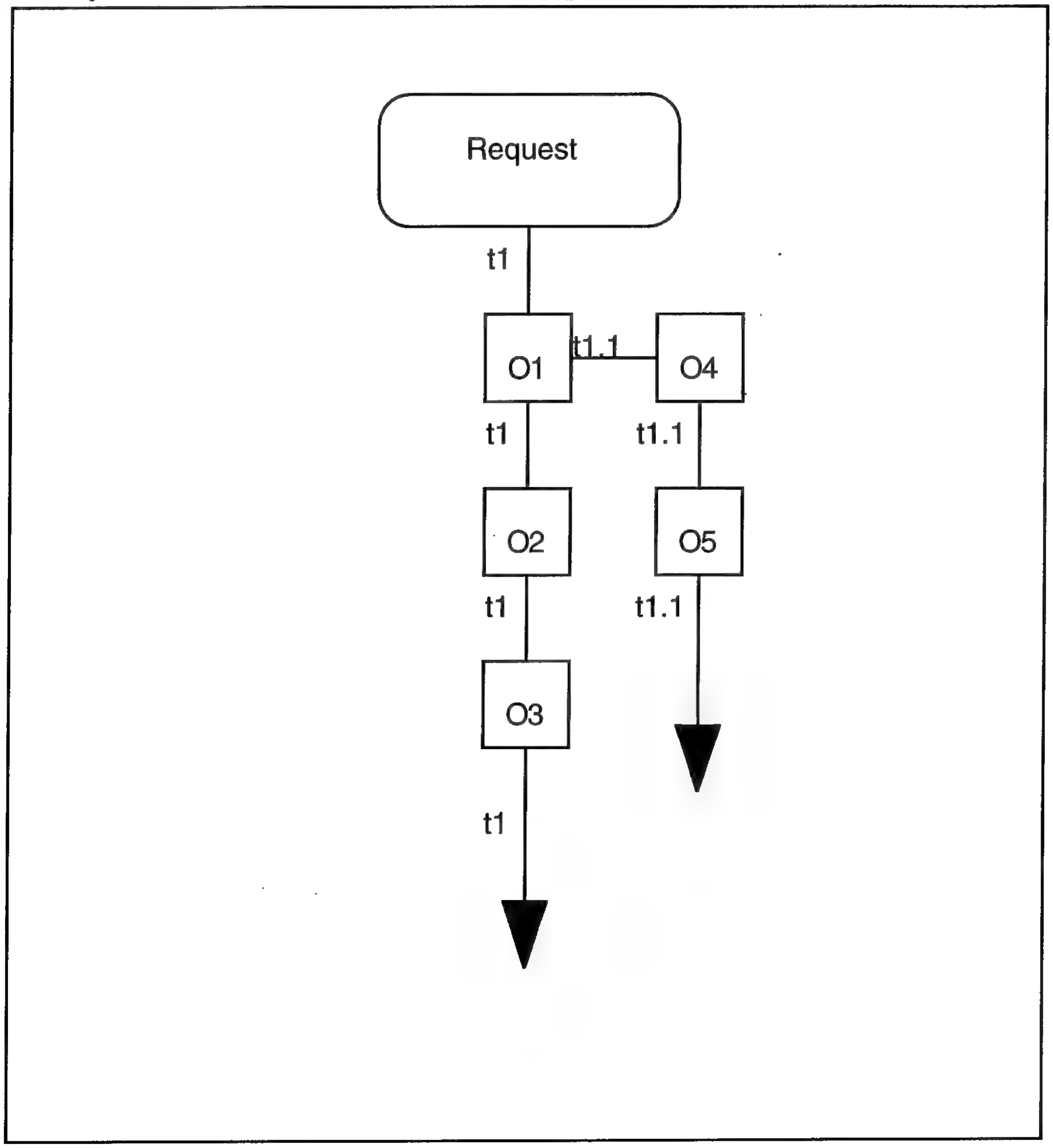

Figure 5. Threads and Sub-Threads

6.7 TELL and COASK

DERIC contains two programming constructs that are not found in ERIC. The two contructs *TELL* and *COASK* allow a programmer to create parallel threads. *TELL* is similar to the *ASK* construct in ERIC, but while *ASK* waits for the object to return a value to the caller, *TELL* simply allows the programmer to make additions or modifications without waiting for return values. For example:

```
(tell artist draw a circle at 50 50)
```

would cause the Artist object to draw a circle at the coordinates (50, 50) without returning any value. *TELL* always spawns a new thread or subthread.

The *COASK* construct can generate several threads at once. *COASK* lets the programmer send messages to several objects in parallel (any combination of different objects and messages). *COASK* will only return when the spawned threads return.

```
(coask   ((list  <objects>)  <message>)
         ((list  <objects>)  <message>)
         . . .)
```

For example:

```
(coask  ((list Radar1 Radar2 Radar3) sweep)
        ((list Sam1 Sam2 Sam3) fire))
```

would send each of the three objects Radar1, Radar2, and Radar3 the message "sweep", while sending the message "fire" to Sam1, Sam2, and Sam3. These events would occur in parallel. The *COASK* will then wait until all of these spawned threads return, and then return the lists of values that each separate object request returned.

6.8 Deadlock

ERIC is a sequential language, and therefore deadlock is impossible. In DERIC, however, deadlock is a very real possibility. In order to discuss how deadlock occurs and is dealt with in DERIC we introduce the terms *externally atomic* and *internally atomic*. An object is externally atomic if it completes all previously requested behaviors before it completes any new behaviors, while an object is considered internally atomic if its internal state cannot be modified while it is performing object requests. An object can be both externally atomic and internally atomic. It is possible to violate an object's atomicity, but each thread must have an identity describing its permissions for violating that object's atomicity to be able to do so.

It is important in simulations to ensure that objects complete behaviors in the correct order. This is not a problem in ERIC because there is only one thread. In DERIC, casuality (maintaining the correct order of events) is ensured by threads "seizing" objects and blocking out all new requests. This can cause problems because a seized object can become blocked, which can lead to deadlock. Deadlock itself occurs when a cycle of seized objects make requests of other seized objects within the cycle, bringing the simulation to a standstill.

6.8.1 Deadlock Avoidance

The **atomicity-level** special variable controls what type of threads can seize an object while the object is in a "waiting" state. The thread that put an object in a waiting state can always reseize it, except for level 5. There are six levels to the *atomicity-level* variable, from 0 to 5. Level 0 is the least safe, allowing any threads to seize an object and possibly cause atomicity violations, while level 5 is the most safe, allowing absolutely no threads to seize the object and possibly deadlock the simulation. The default level is 1, which allows safe threads (defined below) and subthreads to make requests of objects. Often using these settings is the only way to avoid deadlock.

(allow-all-phantoms)	Level 0-allows all threads to seize the object
(allow-safe-and-my-phantoms)	Level 1-allows safe threads and subthreads
(allow-safe-phantoms)	Level 2-allows only safe threads
(allow-my-phantoms)	Level 3-allows only subthreads
(allow-my-safe-phantoms)	Level 4-allows only safe subthreads
(allow-no-phantoms)	Level 5-allows no threads or subthreads

These forms are used within the execution of a behavior to automatically set the *atomicity-level*. For example:

```
(ask George when receiving (swim upstream)
    (ask myself to enter current)
    (allow-safe-phantoms)
    (ask myself to swim for 1 minute)
    (allow no phantoms)
    (ask myself to leave current))
```

We can ask *George* to swim upstream, and then safely ask him (while he is swimming):

```
(ask George recall your depth)
```

and receive an answer. All recall messages are safe and, although they violate an object's external atomicity, its internal atomicity will be unaffected. The concept of safe threads and subthreads is very important. It means that the threads are unable to violate an object's internal atomicity, which could affect the outcome of the simulation. Level 0, *allow-all-phantoms*, cannot guarantee internal atomicity. The message:

(ask class when receiving (<pattern>) safely
{<action>*})

where <pattern> is any message and <action> is zero or more Lisp forms that are to be evaluated when the behavior is envoked, will create safe behaviors. An example of a safe behavior is the "read only" behaviors for "recall" messages, such as:

```
(ask fred recall your print-name)
```

The above will allow *fred* to respond but does not violate his internal atomicity or place the simulation in peril of deadlock. When a user sends a safe messages to an object, the user is assured that the blocked object's internal atomicity will be maintained.

However, allowing an object's atomicity to be violated, even when its "safe," may still lead to simulation correctness problems. If, for example, we asked *Fred* to recall his location just before he updated it, our simulation may run incorrectly (versus running the simulation sequentially). One must take great care in violating and object's atomicity.

6.9 Debugging

Another useful tool for debugging is the function

(print-em)

which displays information about the various threads in the environment. This enables the user to manually detect deadlock by spotting locked threads. Deadlocked threads can currently only be reconciled by using the following commands:

(reset-pipes)
(reset-statuses)
(reset-queues)

6.10 Other ERIC/DERIC Differences

Other than the major differences between ERIC and DERIC already described in this chapter, there are a few other minor differences that should be noted.

There are three messages defined for *something* in ERIC that are not in DERIC. These are the "record" message mentioned above, and the two "make plist" messages. There are also several messages defined for the system objects in DERIC that were not in ERIC. Many of these have already been discussed in this chapter; the rest are listed in the section Behaviors Defined for System Objects at the end of this manual. Note that a number of these messages are not documents in this manual, but there use should be obvious from the message.

One type of message defined for *something* only in DERIC are for evaluating Lisp forms in either the local or remote environment. These are:

(ask <object> eval <forms>)
(ask <object> globally eval <forms>)

The first will evaluate the Lisp forms <forms> in the home environment of <object>. The second will evaluate <forms> in all known environments. One useful example is:

```
(ask something globally eval (setq old-clock clock clock %clock))
```

Further Readings

If you are interested in reading more about some of the topics covered in this report, the following references may be of interest.

Dependencies, Demons, and Graphical Interfaces in the Ross Language, Rand Note N-2589 DARPA, The Rand Corporation, 1988.

MacArthur, David, and Philip Klahr, *The ROSS Language Manual,* Rand Note N-1854-AF, The Rand Corporation, September, 1982.

MacArthur, David, Philip Klahr, and S. Narain, *ROSS: An Object-Oriented Language for Constructing Simulations,* Technical Note R-3160-AF, RAND Corporation, Santa Monica, California, December 1984.

Steele Jr., Guy L., *Common Lisp: The Language*, Digital Press, 1984.

Stefik, Mark, and Daniel G. Bobrow, *Object-Oriented Programming: Themes and Variations,* The AI Magazine, pp. 40-62, Winter 1986.

Behaviors Defined for System Objects

Behavior	Page
Something	
add >x to your list of >attribute	21
after receiving >pattern +actions	18
after receiving >pattern safely +actions	(D)38
announce yourself to +environment	(D)33
before receiving >pattern +actions	18
before receiving >pattern safely +actions	(D)38
delete >x from your list of >attribute	21
eval >form	(D)39
forget your local after daemons matching >pattern	18
forget your local before daemons matching >pattern	18
forget your local behaviors matching >pattern	18
forget your plans after time >x	27
forget your plans at time >x	27
forget your plans before time >x	27
forget your plans between >time1 and >time2	27
forget your plans matching >pattern	27
globally eval >form	(D)39
initialize yourself	17
kill all your instances	18
kill all your offspring	(D)ND
kill yourself	18
make instance >ob	17
make instance >ob with +attributes	17
make plist instance >ob	(E)17
make plist instance >ob with +attributes	(E)17
migrate yourself to +environment	(D)33
print your >attribute	19
print your attributes	19
print your local messages	19
print your messages matching >pattern	19
print your messages	19
print your self	18
print yourself	18
recall the >attribute for your class	19
recall your >attribute	19
recall your after daemons	(D)ND
recall your attributes	20
recall your before daemons	(D)ND
recall your descendants	20
recall your instances	20
recall your local messages	20
recall your messages matching >pattern	20
recall your messages	20
recall your primary behaviors	(D)ND
recall your subclasses	20
recall your superclasses	20
record all	(D)ND
record your messages matching >pattern	22
record your messages matching >pattern to >stream	(E)22
record your messages matching >pattern to >my-recorder	(D)34
reset your stubs	(D)ND
set the >attribute for your class to >value	21
set your >attribute to >value	21
to ask each of your >attribute to +action	21

Behavior	Page
to ask your >attribute to +action	21
to ask your instances to +action	21
trace all	22
trace your messages matching >pattern	22
trace your messages matching >pattern to >my-tracer	(D)34
unrecord all	(D)ND
unrecord your messages matching >pattern	22
untrace all	22
untrace your messages matching >pattern	22
when receiving >pattern +actions	18
when receiving >pattern safely +actions	38
Clock	
cancel all events	28
run for >x hours	25
run for >x minutes	25
run to completion	23
set time to +x	25
tick >n times	23
tick	23
to schedule >object to >action at >x	24
to schedule >object to >action at time >x	24
to schedule >object to >action in >x hour	24
to schedule >object to >action in >x hours >y minutes and >z seconds	24
to schedule >object to >action in >x hours and >y minutes	24
to schedule >object to >action in >x hours	24
to schedule >object to >action in >x minute	24
to schedule >object to >action in >x minutes	24
to schedule >object to >action in >x second	24
to schedule >object to >action in >x seconds	24
to schedule >object to >action in >x time units	24
what date is it	25
what time is it	25
Slave-Clock (D)	
cancel all local events	ND
it is now >time	ND
recall the slave-clock of >environment	ND
Master-Clock (D)	
>clock is active	ND
>clock is inactive	ND
Object-Maker (D)	
make >name a >parent	32
make >name a >parent with +attributes	33
make >name a subclass of +parent	31
make >name a subclass of +parents with +attributes	32
make >name an >parent	33
make >name an >parent with +attributes	32
make an anonymous >parent	ND
make an anonymous >parent with +attributes	ND
recall the object-maker of >environment	33
remake >name a >parent with +attributes	32

Behavior Page

remake >name a subclass of +new-parents with +attributes . 32
remake >name an >parent with +attributes 32

Tracer (D)

recall the tracer of >environment 34
set your tracing-stream to !stream 34
set your recording-stream to !stream 34

(D) - In DERIC only
(E) - In ERIC only
ND - Not documented

Index

%Clock 36
atomicity-level 38
accessor functions 29
add 21
after daemons 13
alive 26
announce 33
attributes 19
before daemons 13
behaviors 3, 10,18
cancel 28
class 19
class-object? 8, 29
classes 3
clock 23
CLOS 29
COASK 37
consolidate-my-environment 32
continue-passing 15
daemons 13, 14, 18
date 25
dead 26
dead-object? 29
deadlock 38
define-class 4, 32
delete 21
DERIC 31
descendants 20
event queue 25
event-list 23
externally atomic 38
file 22
forget 18, 27
get-plan-action 27
get-plan-scheduler 27
get-plan-time 27
inheritance 6
initialize 17
instance-object? 29
instances 20
internally atomic 38
kill 18, 6
live-object? 29
make instance 17
make instance message 8
make plist instance >ob 17
make-object 9
Master-Slave Clock 35
messages 10, 11, 19, 20
methods 16
migrate 34
new-environment 32
object requests 36
object-get 29
object-maker 32
object-put 29
object? 29
pattern matching 11, 12, 29
plans 27
plist 17
predefined attributes 8
predicates 29
print 18, 19
recall 8, 19
record 22, 34
run 25
schedule 24, 26, 27
set 21
set time 25
simtime 23, 24
something 4
subclasses 20
superclasses 20
TELL 37
threads of control 36
tick 23
ticksize 23
time 25
trace 22, 34
Tracer 34
Universal Time 24, 25
unrecord 22
untrace 22
wrappers 15

Rome Laboratory

Customer Satisfaction Survey

RL-TR-________

Please complete this survey, and mail to RL/IMPS, 26 Electronic Pky, Griffiss AFB NY 13441-4514. Your assessment and feedback regarding this technical report will allow Rome Laboratory to have a vehicle to continuously improve our methods of research, publication, and customer satisfaction. Your assistance is greatly appreciated.
Thank You

Organization Name:_______________________________________(Optional)

Organization POC: _______________________________________(Optional)

Address:____

1. On a scale of 1 to 5 how would you rate the technology developed under this research?

5-Extremely Useful   1-Not Useful/Wasteful

Rating_____

Please use the space below to comment on your rating. Please suggest improvements. Use the back of this sheet if necessary.

2. Do any specific areas of the report stand out as exceptional?

Yes___ No______

If yes, please identify the area(s), and comment on what aspects make them "stand out."

3. Do any specific areas of the report stand out as inferior?

Yes___ No___

If yes, please identify the area(s), and comment on what aspects make them "stand out."

4. Please utilize the space below to comment on any other aspects of the report. Comments on both technical content and reporting format are desired.